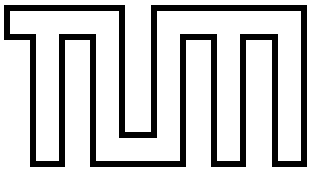


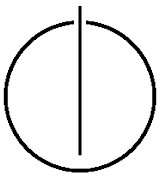
FAKULTÄT FÜR INFORMATIK

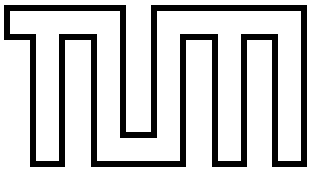
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Smart Grid-Demonstrator: Office Employee Module

Florian Sellmayr





FAKULTÄT FÜR INFORMATIK

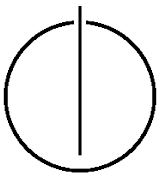
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Smart Grid-Demonstrator: Office Employee Module

Smart Grid-Demonstrator: Office Employee Modul

Author:	Florian Sellmayr
Supervisor:	PD Dr. rer. nat. Bernhard Schätz
Advisor:	Dagmar Koß
Date:	September 26, 2011



Disclaimer

This thesis is the result of a shared development project conducted by Steffen Bauereiß and Florian Sellmayr. Even though both focused on different perspectives on the subject, the resulting architecture cannot entirely be seperated into two different pieces of work.

The project will therefore be described in one document that is being submitted to the university as two seperate but contentwise identical theses to which each author contributed the parts that fit best to his previous focus of work.

To distinguish the two authors work, the text is annotated (usually at the beginning of one section) with notes marking the beginning of one authors text. The text spanning to the next annotation is therefore to be considered as the respective authors work.

Any parts of the text annotated with a name different than the undersigned should therefore be considered as "declared resources" in the sense of the statement below.

I assure the single handed composition of this Bachelor's Thesis
only supported by declared resources.

München, September 26, 2011 Florian Sellmayr

Abstract

This thesis introduces an IT-system controlling a Smart Micro Grid - the Living Lab demonstrator. The demonstrator is set up in an office environment consisting of a set of home automation devices and external systems. A Service Oriented Architecture was realized following several design principles for a highly extendable distributed system. The whole system is designed to control all connected devices in a way that puts the ideas of the Smart Grid which are focused on energy efficiency and communication between different participants of the grid into practice and will be the base for research projects approaching the subject “Smart Grid” and “Smart Micro Grid” from multiple points of view.

Contents

Outline

Part I: Introduction and Problem Statement

CHAPTER 1: INTRODUCTION

This chapter will give a brief introduction into the general idea of Smart Grids and Smart Micro Grids in particular.

CHAPTER 2: PROBLEM STATEMENT

After discussing the general setting in Chapter 1, this chapter will introduce the basic problem this thesis will cover, describe the basic requirements the system to be developed shall fulfill and highlight the constraints important in design.
It will also briefly explain the hardware landscape the system will operate in.

Part II: System Architecture

CHAPTER 3: ARCHITECTURAL PRINCIPLES

Before diving into the details of the architecture proposed for the system, this chapter gives a more general overview about the underlying principles and explains some inherent terms that will be used in the remainder of this thesis.

CHAPTER 4: ARCHITECTURE DESCRIPTION

This chapter describes the core architecture the system is built on.

CHAPTER 5: COMMUNICATION FRAMEWORK

This chapter will give a more detailed information on the principles the architecture defines for communication between components. It will also explain the general abstraction defined for transmitting sensor information and commands.

CHAPTER 6: ALTERNATIVES - THE ROADS NOT TAKEN

This chapter will describe some architectural ideas that were evaluated during the design of the system and the reasons for discarding them.

Part III: Components

This part gives a detailed explanation of the design, implementation and usage of the main components developed as part of this thesis.

Part IV: Further Research and Conclusion

CHAPTER 11: FURTHER RESEARCH

This part describes possible ideas for future work on the demonstrator, shortcomings of the current system and issues not covered in this thesis that may prove to be interesting areas of research for the future.

Part I.

Introduction and Problem Statement

1. Introduction

Author: Steffen Bauereiß

Decentralisation of power-production, “prosumers”, Smart Grid - these are just a few buzzwords found in current newspaper articles and speeches.

At the moment, more than ever, energy production is in the focus of the public and renewable energies are rising. This shift in energy production brings a decentralisation of the power grid with smaller power production units such as wind engines or solar panels all over the country. But not only decentralisation is of concern - a shift of the energy system towards the consumers becoming so-called “prosumers” producing their own electricity can also be seen. This leads to new requirements towards the power grid and results in the ideas of the so-called “Smart Grid”.

This Smart Grid can be defined by both a power and information grid between the prosumers. The basic idea behind it can be explained in a simple way. Each participant of the Smart Grid can decide on producing his own electricity or using the grid as power source based on factors as cost, the weather or any other reason. Information exchange between the participants helps to keep track of energy consumption and needs and helps to achieve the main goal of higher energy-efficiency. But prosumers and small energy production units are not the only power sources of the Smart Grid. Large power plants still have a right to exist and are necessary to cover large demands. From the Smart Grids point of view there is no distinction between a prosumer, small energy production unit or a large power plant. They are all connected to the grid in the same way. Groupings of prosumers or energy production units within the Smart Grid result in virtual power plants. Again from the view of a Smart Grid there is no difference between a virtual and a real power plant. Such groupings can make sense based on geographical proximity or other similarities.

Besides power production the power consumption shall be seen in a completely different way as well. An intelligent system can take control of power production and consumption in the grid. Information exchange between all participants helps to avoid peaks and shortages in power demand. Therefore information has to be gathered at the level of each producer and consumer - each prosumer. An IT-System has to be deployed at each participant of the Smart Grid. But these IT-Systems cannot gather information only, in fact they can be intelligent systems taking control of power consumption due to devices such as lights and other building equipment, electric devices, a backup battery system, and others. These systems shall have the ability to power off unused consumers and adjust the buildings production/consumption of electricity to both the prosumers and the grids present needs. The overall target of controlling all devices in a central IT-System is to achieve higher energy-efficiency.

With these home automation devices all connected to a single system a variety of ser-

vices like alarming, remote controlling, invoice for power consumption and others can be provided as well. Furthermore in this small environment of single devices consuming or producing power, connected to both the power grid and a system that controls them the term Smart Micro Grid can be defined. Each prosumer of the Smart Grid controls his own Smart Micro Grid of devices and manages them in the most energy-efficient way. IT-Systems like the Living Lab demonstrator developed in the scope of this thesis aim at controlling a Smart Micro Grid consisting of a variety of home automation devices and other IT-Systems to connect to as well as other participants of the Smart Grid to exchange data with. They are a new field of research trying to put the ideas of the Smart Grid into practice.

2. Problem Statement

2.1. Basic Problem to solve

Author: Steffen Bauereiß

In order to test, evaluate and demonstrate different approaches to build the software backbone of a Smart Micro Grid, a basic system architecture for the Micro Grid Living Lab has to be developed. The idea behind the first stage of the Living Lab demonstrator is developing a sample-environment representing one participant of the Smart Grid. In this context a variety of requirements for the Living Lab demonstrator can be formulated. The Smart Grid combines both the control of power production and consumption and interaction between multiple prosumers. For the Smart Micro Grid Living Lab the very same ideas need to be considererd: controlling devices and connecting to other systems.

Therefore the first idea is that some kind of decision algorithm has to take control over all connected devices and put the ideas of the Smart Grid into practice. For this automated control, the system needs to collect information about every connected device and external system. The second requirement is of course interaction between the different systems connected via the Smart Grid and located at different levels of the grid. A third requirement to the system is configurability. In addition to the requirements described above, each stakeholder should have the ability to configure different parts of the system according to his needs.

On a very basic level one could therefore talk of these requirements: Connecting actors and sensors to the Living Lab demonstrator as well as working with a decision algorithm and offering the ability to collect and exchange data with other systems of the Smart Grid.

2.2. Current System Environment

Author: Steffen Bauereiß

In the first development stage of the Living Lab demonstrator the system environment is broken down to a simple environment consisting of a set of home automation devices available on the local network. Nevertheless connecting other systems has always been a requirement for the architecture but no particular systems are considered in the first stage of the system-environment. Later systems like a central recording of power consumption for invoice, systems to control appliances elsewhere, alarming systems and others will be part of the system environment. For this simple environment a short overview of available hardware is provided in the following sections.

2.2.1. EnOcean

For the Micro Grid Living Lab demonstrator EnOcean devices play an important role. Being perfectly suitable for demonstration purposes the focus has so far been on connecting those devices to the system.

The EnOcean product family consists of two categories of devices communicating wirelessly. Sensors providing data reach from simple lightswitches and dimmers to temperature and occupancy sensors. Actors on the other hand, imaginable as small control-boxes are connected to a variety of appliances such as lights, AC, heating or blinds. Actors always act upon messages sent from sensors. These messages are called telegrams. In order to have a specific actor react upon a telegram from the desired sensor, the user has to assign the sensor to the actor using a built-in learn-mode. An EnOcean actor does not have an ID - the only way to talk to such actors is by sending telegrams they understand. Having an actor understand a telegram means the actor has to know the ID inside the telegram (= ID of the sender of the telegram) it receives. That's the purpose of the EnOcean learn-mode and results in the requirement to support this learn-in procedure by the wrapper used for communication with EnOcean devices. Another point to mention is the communication between the system and EnOcean devices. As mentioned before, communication between EnOcean devices is done by sending telegrams. In order to have the Living Lab demonstrator talk to the actors and listen to the sensors an additional device capable of receiving and sending telegrams is necessary. A gateway, in this case "Thermokon STC Ethernet"[1] is connected to the local network and is both capable of receiving each telegram sent by a sensor and sending telegrams to actors.

2.2.2. IPSwitch

The IPswitch is a small device with the functionality of a power tracker. Measuring the power consumption of up to three appliances as well as the environments temperature and providing it to the local network is the only functionality available. Remote-resetting the device is provided as well.

2.2.3. Siemens Pac Sentron and Siemens PowerBridge

The Pac Sentron manufactured by Siemens is a measuring device for a variety of values. For external access this multi-purpose measuring device is connected to the local network and offeres a variety of configuration possibilities. Siemens provides an additional gateway to combine measuring devices and provide easier configuration. This gateway called PowerBridge is capable of running custom code. With the Living Lab demonstrator designed as a distributed system this PowerBridge can be directly integrated into the system by running the software component for communication with Siemens devices on it.

[1] see `http://www.thermokon.de/EN/easysens-31/stcethernet--gateway-264.html` (accessed 25th September 2011)

2.3. Constraints

Author: Florian Sellmayr

A system as envisioned above which has to perform well for a wide variety of different use-cases (possibly including quite a few not anticipated at this point) has to be designed with special care when it should be useful for research for a prolonged period of time.

It therefore makes sense to present the basic constraints and non-functional requirements that govern the design of the system[2].

2.3.1. Flexibility and independence of components

The idea of having a basic system to demonstrate different approaches to Smart Grids, communication protocols and hardware systems requires the ability to quickly integrate, replace or remove software components or secondary hardware systems into the main architecture while also preserving the components ability to interact with each other without any extra effort. A decision-making algorithm should be able to control any device and work with sensor-data from all types of sources without knowing about their implementation details.

Furthermore, the system should be as technology-agnostic as possible, enabling easy integration of foreign components, protocols and libraries, e.g. a webservice-based alarming service or UIs and hardware wrappers running on foreign platforms such as mobile devices or embedded systems.

2.3.2. Scalability

To have significance for real-world applications, the system should also scale well with respect to the amount of data processed, the number of devices handled and should be able to keep latency low for certain operations.

While our initial demonstrator setup covers only very few sensors and actors (for details see section 2.2), a typical Micro Grid can easily contain hundreds or thousands of different devices, depending on the size and type of the building to be covered. The system should therefore be able to easily handle that many devices that can be accessed either in an aggregated form via a gateway device (as it is the case with EnOcean compatible devices) or directly (as it is the case for the IPSwitch).

With the number of devices attached to the system, obviously the size of data to be processed (in terms of sensor information flowing into the system, commands being issued in reaction to new data and possible message exchange between components to gather additional information necessary to perform a task) grows as well.

The system therefore has to be able to deal with such amounts of data to be useful.

In particular, the system latency should be low for high-priority data such as the information that a lightswitch was pressed. In this case, immediate processing and reaction is necessary to provide a good user-experience: While having to wait several seconds for analyzing sensor-data might be acceptable under most circumstances, waiting that long

[2] not all of these factors had an immediate impact on the development of the system-architecture and/or are only relevant for some components

for a lightswitch to react is obviously undesired behavior and could result in acceptance problems on the user-side.

2.3.3. Distributability

A Smart Micro Grid in itself can never be a stand-alone system. It has to connect to a bigger Smart Grid infrastructure and has to integrate a number of different energy consumers, producers and sensors distributed over a large area.

Some of these external systems (e.g. the Siemens PowerBridge) already provide a way to run custom made software-components on the device itself (see section 2.2.3). In terms of efficiency, it would therefore make perfect sense to run the code interfacing with the Micro Grid infrastructure directly on the device. In a future commercial implementation of this kind of infrastructure, having a standardized interface for such systems that are implemented on the device itself will probably be the optimal solution.

Furthermore, it might be advantageous to distribute internal parts of the system, such as data storage or decision making systems over multiple machines.

Thus, the system design has to provide interfaces and a communication infrastructure to enable the system to be distributed over different platforms, physical locations and networks as effortlessly and flexible as possible.

2.3.4. Robustness and self-recovery

As the system to be developed is planned to be the central backbone of all building automation technologies within the building, robustness should be another crucial concern when developing the system.

While the components itself should be reliable, robust and self-recovering, the same issues also have to be addressed with respect to the connection of external components to the system:

External components such as sensors, actors and protocols are adhering to different standards for precision and reliability of data as well as reliability of the devices in general.

Robustness of their integration therefore has to be considered in two different contexts:

Robustness with respect to external systems behavior

The system has to be able to deal with possible flawed input from external devices without affecting the general functions of the system: Failure in one component must not affect the function of components unrelated to the failing component. And for components related to the failing component, effects should be minimized to maintain usability as well as possible.

As far as external systems allow (some hardware setups may require human intervention in some situations), the architecture should support methods to automatically recover from common failures, such as temporary loss of network connectivity to a device.

Robustness with respect to data reliability

Since all sensors have inherent flaws limiting the sensors resolution with respect to time and unit measured, the system must be designed in a way to be able to work with possibly

flawed data.

2.3.5. Energy efficiency

As already mentioned, Smart Grids are a tool to use energy more efficiently. For a Micro Grid infrastructure to be effective it is therefore important for the sytem to keep additional energy-consumption to a minimum as not to counteract the energy-savings achieved.

For a general technology-demonstrator however, this can be considered to be a low-priority requirement that does not trump other considerations in design decisions.

Part II.

System Architecture

3. Architectural Principles

Author: Florian Sellmayr

Before going into the systems architectural details, this chapter will explain in more detail the principles governing the architecture, what they mean and why they were chosen.

3.1. Service Oriented Architecture (SOA)

The main architecture philosophy underlying the architecture to be presented is called *Service Oriented Architecture* or in short, *SOA*.

Because of the recent hype in this field, there is no precise common definition or understanding on what a SOA actually is. So before getting into the details, lets take a look at two Service Oriented Architecture principles that we find central to the nature of the system to be developed.

These principles are mainly inspired from Dirk Krafzigs Book on *Enterprise SOA* ([1]).

3.1.1. Services

The probably most important component of the architecture philosophy is the **Service**: A service is typically a component within the system that provides a clearly defined, single functionality (such as "long term persistence of sensor data") and encapsulates all logic and data necessary for this task.

It is not to be confused with a technical service (e.g. access to a database): A SOA should be totally technology-agnostic, leaving technical details to the service implementation.

To access data or logic represented by a service, it provides a high-level service-contract and interface that provides all information necessary to work with the service. Again, this service interface is written with application domain principles in mind, not technical ones (e.g.*"give me the most recent sensor value for sensor42"*, not *"execute SELECT ..."*).

This makes it easy to interact with services without knowledge about their internal structure and to replace a service implementation without disturbing the rest of the system.

Services can provide basic functionalities (such as providing historic sensor data or information about relationships between devices) or consume other services to provide more high-level functionalities (such as analysis of historic sensor data with respect to their relationships, therefore consuming the two services mentioned above to produce more high-level information).

(For more details see [1], Part I, Chapter 5: *"Services as Building Blocks"*. It also offers a fine grained naming scheme for different types of services. Since it is focused on the context of Enterprise-Applications these names are not used in the context of our system.)

3.1.2. The Service Bus

As previously stated, services may exchange information to provide their functionality. Thus, the architecture must define a way to make this communication happen.

The preferred architecture pattern in a SOA is the *Service Bus*. It provides a common interface to exchange information between services and offers one simple communication endpoint for all communication within the architecture. To send messages to a service, only the physical location (typically the IP-Adress) of the Service Bus and the services name is needed. The Service Bus is responsible for discovery and routing of the messages. If needed, it can also provide message transformation, QOS, security, filtering or logging mechanisms.

This way, services can be developed using a variety of technologies, programming languages or vendors without having to deal with the usual integration problems in heterogenous software landscapes.

For more details see [1], Part I, Chapter 9: *"Infrastructure of the Service Bus"*.

3.2. Event-driven communication

When designing the flow of data inside the system, two principles come to mind: **push-based** and **pull-based** communication (push and pull seen from the perspective of a component receiving data).

In the first case, new information (e.g. sensor data) will be fed into the system as it occurrs, leaving it to the receiving components (e.g. persistence or decision algorithms) to act upon such new information as it arrives.

The second option would mean storing all information within the components responsible for retrieving such information from the devices and exposing them on request.

While the latter option might be cleaner in the sense of "ownership of data" (every component is responsible for all information about the devices it manages), the first option (in our case called "Event-driven communication") was chosen for this system[1].

Event-driven means that all actions within the system are triggered asynchronously by events such as the pressing of a button (which initiates an event message which in turn initiates processing by other components which may in turn initiate other actions[2]), a timer or simply some action on the user interface.

This kind of event-driven communication model allows to build a system with minimal latency (since events are being sent and processed as they occur) and reduced load on the communication infrastructure (since polling is unnecessary), which again improves latency and decreases the necessary computing power, therefore improving the infrastructures energy-usage footprint.

Messages that are triggered in this way can generally be divided into the following three groups:

- **Measurement Events**[3]: Messages coming from a device to indicate new information on the state of the physical world. This can be classical sensor information (*"bright-*

[1] for a more detailed analysis of the pull-principle see section 6.1

[2] for a more detailed example of such a flow of information see section 4.4

[3] from now on, often simply referred to as "Events"

ness is 10.000 LUX", *"current power usage 80W"*), information on the state of devices controlled by the system (*"Light 42 now switched on"*, *"Blinds at 25%"*) or internal management information (*"Found new Device"*). These events are typically used by decision-making algorithms to determine what happened and what actions to take.

Note that no contract was defined with respect to the semantical meaning of a new event. Devices may choose to inform the system periodically, when changes in the measured value are detected or when this value was above a threshold. It remains the responsibility of components consuming events to handle these cases accordingly.

- **Commands**: These are messages instructing a device to perform a specific action (*"Switch light off"*, *"Charge batteries now"*). These commands are usually sent from decision-making algorithms or the UI to components managing devices.

  Note that no guarantees are given with respect to conflicting or redundant commands. While handling of redundant commands is the responsibilty of the component receiving the command as far as possible, some types of redundant and all conflicting messages have to be handled and resolved in a specialized, probably complex way that is out of scope of this thesis[4].

- **Remote Procedure Calls** These messages are used for general exchange of information between different components (*"store association between device A and B"*, *"give me all data for sensor 42 recorded in the last 24 hours"*).

To make this communication model easy to extend, these messages follow a clearly defined structure: Signatures of individual components external interfaces (or RPC-signatures) are explained together with the individual components in part III whereas the structure of events and commands is explained in more detail in section 5.2.

3.3. Component Framework

For a system of this size, with flexibility as a major requirement, structuring the system in some way is of paramount importance.

For this purpose, parts of the system are divided into different components. These components are independent parts of the system that have a clearly defined purpose, offer distinct functionalities and are solely responsible for their own data in compliance with the SOA philosophy.

They may also depend on clearly defined services from other components and communicate with them via the communication principles described in section 3.2.

Furthermore, the component framework supports dynamic starting, stopping and replacement of components at runtime, allowing quick deployment of new features or software releases on the fly, without disturbing the systems operation as a whole.

To meet these requirements, OSGi was chosen as the principle runtime environment providing the component framwork.

[4] for more thoughts about this see section 11.3

As the underlying framework for the Eclipse Platform as well a number of other applications[5] it is supported by a vibrant and stable community that includes industry heavyweights such as IBM, Oracle or Siemens and Open Source Organisations such as the Eclipse Foundation[6].

The framwork is build around components (called *"Bundles"*) that are standard Java JAR-Files with additional Meta-Information (called *"Manifest-File"*). While they can be used to define dependencies on a specific bundle, the more common way is to define dependencies on Java-Packages to import and leave it to the framework to find a bundle that exports these packages.

Bundles are deployed into an OSGi-container (in this system Eclipse Equinox is used) which takes care of dependency resolution, classloading and offers management-interfaces to start, stop, install, uninstall or replace bundles at runtime[7]

[5] for an overview see `http://www.osgi.org/Markets/HomePage` (accessed 25th September 2011)
[6] for a broader overview see `http://www.osgi.org/About/Members` (accessed 25th September 2011)
[7] for more detailed information on OSGi, see [4].

4. Architecture Description

Author: Steffen Bauereiß

All principles described in chapter 3 can be found in the final architecture for the Living Lab demonstrator. Figure 1 shows a number of components attached to a Service-Bus as communication-channel (see section 3.1.2). Implemented as an OSGi-Bundle each component is exposed as a service to meet the architecture principles.

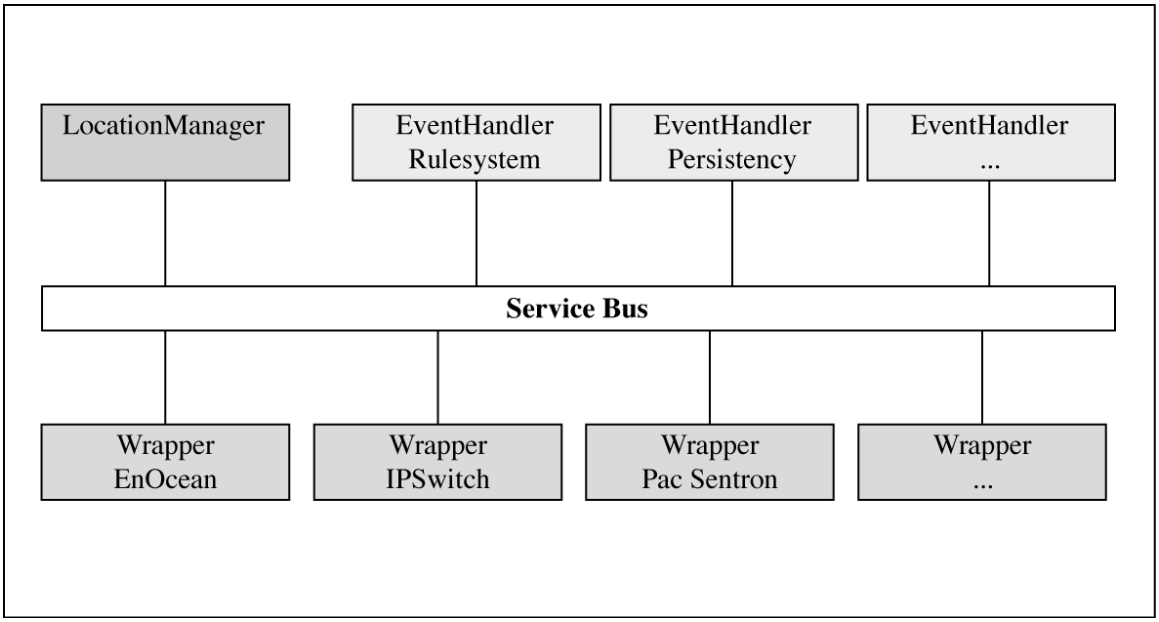


Figure 1.: Living Lab Demonstrator Architecture

4.1. Wrapper

A wrapper used in the Living Lab demonstrator always represents physical devices by connecting to both these devices and the Service-Bus of the system. The main task of a wrapper is to translate between device-specific protocols and the event/command-based communication used in the system. A wrapper usually is a source of events for the system by forwarding incoming device data as events to the system. At the same time by implementing the wrapper-interface it is ensured a wrapper understands all types of commands the system can send to wrappers. A more detailed description of a wrapper implementation can be found in chapter 9. The design of a wrapper is completely left to the programmer as long as he follows the requirements mentioned. Therefore a wrapper can

be designed to control a single device only or in most cases to control many devices. Controlling many devices means the wrapper has to work with internal IDs for the devices in order to control a specific one when necessary. Each event created by the wrapper contains data and the internal ID of the device the data originated from. The usage of internal IDs in the Living Lab demonstrator always follows the scheme concatenating a queue-name and the internal device-ID in the following way:

queue-name?internal-device-ID

This concatenated ID always enables the system to address a specific wrapper and a specific device controlled by this wrapper. The queue-name can be imagined as a ID for a wrapper. An example ID like *enOceanWrapper1?001244* would address device *001244* of *enOceanWrapper1*. A more detailed description of queues and their purpose can be found in section 5.1.

4.2. EventHandler

Seen from the communication point-of-view EventHandlers are the opposite of wrappers - they act as event-receivers. Each event sent is redirected to all EventHandlers. The idea behind the EventHandler is simple: Each data-processing component of the system can implement the EventHandler-interface and by doing so receives all events. A component implementing this interface can still implement other interfaces or create events as a result of other events. As indicated by "..." in figure 1 the system supports multiple EventHandlers. The two components in the Living Lab demonstrator currently implementing the EventHandler-interface in order to receive events are Rulesystem (see chapter 7) and Persistency. They work with these events and either use a rulebase to check whether an event requires an action or simply log them to a database.

4.3. LocationManager

The last component shown in figure 1 is the LocationManager. Again a detailed description of the LocationManager can be found in chapter 8. This component is neither wrapper nor EventHandler. It's a component used for storing data necessary for controlling the system or the Rulesystem checking for device-associations. Entries for each device and connections between devices are stored here.

4.4. Example

While part III gives a very detailed description of the components described above with a focus on their implementation and behaviour, the following section shall help to understand the basics of how they work together.

By following the example scenario of pressing a lightswitch, resulting in a light being turned on all components will play a role, each described with a few sentences.

First of all the lightswitch and light need to be known to the system. In particular a wrapper for the devices needs to be running in the system. Assuming both the lightswitch (LS) and light (L) are EnOcean devices one EnOcean-wrapper (WR) is enough. LS is source

of some kind of message. When pressed this message is generated and redirected to WR. The wrapper can then translate the EnOcean-message into an event used within the system.

This event is received by all EventHandlers. In this scenario, the persistency component updates the database with a log-entry and the Rulesystem (RS) processes the event. A rule tells the system to turn on a light upon receiving the event *LS was pressed*. By sending a request to LocationManager the RS learns *LS is connected to L*. In order to do so the RS sends a command to the EnOcean-wrapper. The command can be imagined as a *turn on L!*-message.

WR receives the command to turn on L and creates an EnOcean-message that is sent to light L. The action can immediately be seen by looking at L that turns on.

5. Communication Framework

Author: Florian Sellmayr

Since communication between different components is the central infrastructure within the system, great effort was taken in selecting a communication framework that meets the demands of the architecture.

To produce a good user-experience with a large number of users, components and devices, the framework would have to be highly scalable with respect to the number of communication endpoints, message throughput and latency.

Since events are supposed to be distributed to a flexible number of components, a publish-subscribe mechanism is of paramount importance as well.

5.1. Implementation Details

5.1.1. JMS specification

After several other attempts (see chapter 6), SOAP over JMS turned out to be the best match for the requirements mentioned above.

JMS, or Java Messaging Service, is a standardized[1] interface for message passing infrastructure in the Java world. It provides a general communication layer based on the concept of unidirectional, ordered communication channels called **Queues** and **Topics**.

Queues are used for one-to-one communication between two components, e.g. to transmit commands from the Rulesystem to one of the wrappers. In case a reply is needed, a second "reply-queue" is used.

Topics on the other hand are providing a many-to-many communication channel based on a publish-subscribe mechanism. Any number of endpoints can subscribe to a topic to which messages can be sent from any number of sources. This mechanism is used to transmit events from wrappers (the event sources) to the subscribing EventHandlers.

Of course, m:n-communication provided by the *topics*-mechanism could also be used for 1:1-communication (i.e. *queues*) just as well as an m:n-communication could be provided by a component that handles several 1:1 communications.

Splitting up these kinds of communication however allows easier development since there is a clear semantical difference between messages that are intended for a specific, well known recipient and messages that are intended for a wide audience. For example, interfaces for topics have to be designed more carefully since recipients may not be known at development time or come from a number of different vendors.

[1] see JSR 914

Some benchmark scenarios[2] suggest however, that topics have better performance characteristics compared to queues.

The design decision to use them even though topics could provide 1:1 communication as well should therefore be questioned in case performance problems become evident at some point. The changes necessary in code would probably be very limited. However, preliminary benchmarks on a basic prototype setup showed no immediate problems in typical usage scenarios.

The second option, creating a component to provide m:n-communication in a 1:1-communication framework would have allowed to consider a much wider range of possible communication infrastructure options such as a standard webservices approach. It was dropped however due to the considerable effort of developing (i.e. "reinventing the wheel") such a component in a way that provides the necessary performance.

5.1.2. JMS Provider

As JMS provider, i.e. the software that implements the JMS standard and provides the low-level management and communication infrastructure, *Apache ActiveMQ* was chosen.

As an Apache project it is a high quality, low cost solution with a stable, vibrant community and integrates very well with other Apache components used in this project.

Furthermore, it is highly flexible and already provides bindings for a large number of programming languages (such as C(++), Javascript, Python) and Protocols (such as REST, XMPP, AMQP)[3]

5.1.3. SOAP over JMS

While JMS provides a very good starting point for low-level communication, it defines no logic whatsoever to handle message content, let alone specify methods for interface definitions or RPC that make it easy for developers to develop new components without in-depth knowledge in JMS or message encoding.

These aspects are very well addressed in the webservice-world (an option that was also previously explored (see chapter 6)).

SOAP over JMS, currently pending standardization as a W3C Working Draft[4] is therefore used as a way to combine both worlds.

While JMS is used in the backend, a webservices framework (*Apache CXF* in this case) provides the translation between high-level Java objects and method-calls and low level JMS-messages.

The workflow therefore resembles typical Webservice-Development. An interface is defined in a platform indepenent WSDL-file from which Java classes and Interfaces are generated[5].

[2] e.g. see `http://fusesource.com/docs/collateral/ActiveMQ%20Performance.pdf` section 6.4 (accessed 25th September 2011)

[3] for a full list of cross-language clients see `https://activemq.apache.org/cross-language-clients.html` (accessed 25th September 2011)

[4] see `http://www.w3.org/TR/soapjms/` (accessed 25th September 2011)

[5] since WSDL and SOAP are not exclusive to the Java world, this approach also allows developers to create components with programming languages other than Java. The only requirement is the ability to communicate with the JMS provider (which should be possible either directly or indirectly under almost all

Java-Classes that want to provide services implement the respective interface and register themselves with the framework. Classes that want to consume a service retrieve a proxy-object from the framework and work with it's methods[6].

Behind the scenes, method-calls are translated into SOAP-messages, sent to the receiver(s) via JMS where they are again unmarshalled and processed as normal method invocations. Returned values (if needed), are again translated into SOAP-messages, sent through a reply-queue where the framework will unmarshall them and return them back to the caller in the expected Java-type.

5.2. Abstraction

Since the system should be able to handle a variety of different systems and protocols designed for a diversity of requirements, developing an interface to model events and commands is a non-trivial problem. If modeled too generalized, interpreting commands and events will be difficult and ambiguous. On the other hand, any effort to model more than very generalized domain knowledge will certainly lead to overfitting, making it difficult to add new components later in the project.

After analyzing the interfaces and behaviors of the proposed initial system environment (see section 2.2) as examples, we come to the conclusion that these systems share very few distinctive properties except for one very basic principle: All of them either produce or receive values that can be generalized into very simple data-types:

- integer or decimal values (often including a unit) (typically data from temperature sensors, smart meters, instructions for dimming a light or setting a position)
- boolean values (typically light-switches, occupancy sensors, lights)
- events without any semantical information (typically simple buttons, lights that can just be toggled)
- character-strings (typically devices showing text or as a general purpose "emergency" tool for cases not covered by the other categories)

After a series of dry-run tests using example scenarios developed ourselves and by *fortiss* employees to check the soundness and flexibility of this approach, the following basic abstractions were defined for the system.

Wrappers can both send and receive messages with the following content:

- `Double(devId:String, value:double, unit:String)`
- `Boolean(devId:String, value:bool)`
- `Toggle(devId:String)`
- `String(devId:String, value:String)`

forseeable circumstances due to huge set of protocols available in ActiveMQ)

[6]For details on how this is implemented using CXF, see `https://cxf.apache.org/docs/soap-over-jms-10-support.html` (accessed 25th September 2011)

Figure 2.: Interface for Commands to the wrapper

WrapperInterface
+ toggle (internalId:String, delay:int, tag:String) : void + stopLastCmd (internalId:String, delay:int, tag:String) : void + setString (value:String, internalId:String, delay:int, tag:String) : void + setDouble (value:double, unit: String, internalId:String, delay:int, tag:String) : void + setBoolean (value:boolean, internalId:String, delay:int, tag:String) : void

Figure 3.: Interface EventHandler

EventHandler
+ TOPIC_ADDRESS : final String
+ newDeviceEvent (origin:String) : void + stringEvent (origin:String, value:String) : void + doubleEvent (origin:String, value:double, unit:String) : void + toggleEvent (origin:String) : void + booleanEvent (origin:String, value:boolean) : void

Additionally, EventHandlers can be informed about new devices with the `newDeviceEvent(devId:String)` and commands can be cancelled by calling `stopLastCMD(devId:String)`

Commands going to the wrappers also have two more attributes:

- `internalId:String` Allow the adressing of a specific device in case the wrapper is responsible for more than one device (for details see section 4.1)
- `tag:String` Allows the user to attach some semantic meaning to commands, e.g. to be able to cancel a command only if it has a certain tag[7]
- `delay:int` Allows the caller to delay the execution of a command, e.g. to give the user a chance to cancel a command issued by the rulesystem before it is executed[8].

Figures 2 and 3 show the interfaces used for sending commands to wrappers[9] and events to EventHandlers.

[7] This is based on an example-scenario where blinds are closed in case of bright sunlight but the movement is stopped once the brightness value inside the office is below a threshold. Without a tag, this rule would also stop the blinds when they were started by the user which is clearly an undesired behavior.

[8] The example scenario for that is based on a behavior often experienced in TUM's CS-department in Garching where automatic blinds often open and close without prior warning or ability to stop them.

[9] This leaves out the `getDevices()` method which is used to provide Metadata about the wrapper

6. Alternatives - The roads not taken

Author: Steffen Bauereiß

During the inital phase of the project different approaches were evaluated in order to determine the best approaches to meet the architecture principles. Some ideas previously explored and the reasons why they were dismissed are described below.

6.1. Push vs. Pull

In section 3.2 different aspects of push vs. pull approaches are already mentioned. The decision to use push-based approach over the pull-based one is due to the fact that many devices connected to the system already support a push-mechanism. Of course the wrapper could keep this pushed information and wait for pulls from the system. Information from devices not supporting push can be pulled periodically by the Wrapper. But looking at a system with a decision algorithm a push-approach seems to make more sense in terms of whenever the state of a device changes, the system gets informed automatically and most important instantly. With a pull-approach a basic latency to a small degree is inevitable and the push-approach helps to avoid this basic latency. Furthermore all components receiving events (EventHandlers) do not have to implement a pull mechanism.

6.2. Bare Webservices approach

The bare webservies approach was dismissed after a few testruns resulting in two arguments clearly speaking against the approach. First, bare webservices have foreseeable issues in a large scale environment with huge numbers of services registered in the service registry, limiting scalability. This limitation is owed the fact that the service registry used in a bare webservices architecture is not built to handle large numbers of services, especially multiple instances of a service which might become necessary once a certain size of system environment is reached. Second, the absence of multicast in bare webservices, clearly limiting communication-possibilities, is another argument against the approach.

6.3. Fully independent devices

The term fully independent devices stands for the idea to offer one instance of a service for one device. This approach leads to enormous numbers of services in large-scale environments. All test-runs with a reasonable large number of services and different system-load-scenarios led to unsatisfactory delays in message transmission and inacceptable roundtrip-times.

6.4. Eclipse SOA

An alternative to the OSGi/ActiveMQ combination to have a flexible services-architecture is using the prebuilt Eclipse SOA environment or similar SOA-packages from third parties. With the last mentioned being costly and Eclipse SOA no longer being maintained the choice was to build an own Service Oriented Architecture with a similar approach. Additionally the prebuilt packages are not meant to be used with dynamically loaded services and multicast-communication necessary in the Living Lab demonstrator.

Part III.

Components

7. Rulesystem

Author: Florian Sellmayr

As an example for a decision-making algorithm for the system, a component was developed that reacts to events based on a set of predefined rules, such as *"Switch on lights when Brightness is above 30.000LUX"*.

Rules can act to incoming events alone, reason over aggregated sensor values or combine values from different sources. They can also query other components and use their results for reasoning or actions.

The most important such component for the Rulesystem is without doubt the LocationManager (described in chapter 8) that stores relationships between devices and therefore enables the user to define generalized rules such as *"Send a Boolean(true)-Command to all devices associated with button `b1`"*.

The following paragraphs will explain the implementation and usage of this rulesystem.

7.1. Technical Background: JBoss Drools

To provide a powerful rule-engine, *JBoss Drools 5.1.1* was used as the backend library.

It has a wide range of features, ranging from basic reasoning to reasoning over time and Complex Event Processing and is supported by a strong open-source community backed by RedHat.

7.1.1. The Rules-Language

First let's examine the language that is used to define rules for the Drools rule-engine.

This will only give a quick overview over the basics of the rule-engine as needed in the scope of this thesis. Fore a more detailed documentation see [5].

A Rule

A rule in Drools is defined as a combination of a **name**, **conditions** that have to match[1] and a **consequence** that is executed if the condition matches:

```
rule "<some name>"
when
    <conditions>
```

[1] the Drools documentation also uses the term "constraint" where one condition is usually formed by one "object-type constraint" and a set of "field constraints". To simplify things this thesis will only use the term "condition"

```
then
    <consequence>
end
```

Listing 1: A rule skeleton

One condition looks like
`$ev : DoubleEvent($origin : origin == "abc")`. This matches a `DoubleEvent` (which is a Fact-Type[2]) from origin `abc` in the engines knowledge base. The dollar-sign prefixed names are (optional) variable names that can be used to identify the event or one of it's parameters in the body of the rule or in subsequent conditions to match in the same rule.

The **Consequence** can be arbitrary Java code that is executed for every match in the knowledge base.

7.1.2. Globals and Imports

For rules to execute actions outside the rulesytem, e.g. call methods on normal Java-Objects, so called *Globals* are used.

They are defined outside the rules, similarly to member-variables in Java classes: **`global`** `Logger lo`
This defines an SLF4J-Logger variable with the name `logger` that can be used in rule bodys (e.g. `logger.debug("Hello World")`).

When defining the global, only the class-name (`Logger`) was used, without the whole package declaration. To do this, the class has to be imported just like in Java:
`import` `org.slf4j.Logger`.

Obviously, this `logger` variable has to be set somewhere. This is done once the rule is added to the Drools rulebase where a mapping between the logger name and a POJO is defined: `ksession.setGlobal("logger", getLoggerObject());`

For an overview of the globals available in the Rulesystem at this point, see section 7.2.2.

7.2. Implementation

7.2.1. Fact Types

Fact Types available in the Rulesystem can be found in the package `org.fortiss.smartmicrogrid.eventhandler.rulesystem.representations`. Most of them are derived from the events described in section 5.2. Per event there is one Fact-Type that contains the events values. Facts of these type are inserted into the rule-engines knowledge-base whenever a new event is received.

Apart from that, there are two more facts that can be inserted into the KnowledgeBase:

Device

This class is a full representation of a device. It encapsulates all knowledge and functionality for a device that is available to the Rulesystem and is an alternative to rules based on events.

[2] the set of facts available are described in section 7.2.1

It provides command methods (named like the commands described in section 5.2, e.g. `setBoolean(value:boolean)`), event-methods (e.g. `booleanEvent(value:boolean)`) that are responsible to pass events on to the device-representation and getter-methods (e.g. `getBoolValue()`) that return the most recent value for this type.

Running

This fact was necessary to represent the notion of a command in-progress to the rulesystem to write rules that react differently depending on whether or not a command is running at this moment[3].

This fact was hand-tailored for this specific purpose and may not be easily useable in a more generalized setting. When developing a more advanced version of the Rulesystem, a more dynamic approach (for example using Drools' FactTemplate mechanism) may be advisable.

7.2.2. Globals

The following globals are defined per default for use in rules:

logger

A standard SLF4J-Logger. For available methods see `http://www.slf4j.org/apidocs/org/slf4j/Logger.html` (accessed 25th September 2011).

rnd

An object of the standard Java-Random Type. For details see `http://download.oracle.com/javase/6/docs/api/java/util/Random.html` (accessed 25th September 2011)

some

An object of type SomeGlobalObject. It has no purpose within a running system but is used in some tests to be able to check the behavior of rules within the system.

commander

A helper class to make it easy to send commands to devices. It provides convenience methods for both single device-IDs and collections of device-IDs so that results from LocationManager-queries can be directly passed to the commander. See figure 4 for an overview of methods provided by the commander class. All methods have the same semantic meaning as described in section 5.2.

[3]The example scenario were blinds that are operated by a button: After pressing button A, the blinds will move down. Pressing button A again while the blinds are moving to stop the blinds

Figure 4.: The Commander-Class

Commander
+ toggle (devName:String, delay:int, tag:String) : void
+ toggle (devName:Collection<String>, delay:int, tag:String) : void
+ stopLastCmd (devName:String, delay:int, tag:String) : void
+ stopLastCmd (devName:Collection<String>, delay:int, tag:String) : void
+ setString (devName:String, value:String, delay:int, tag:String) : void
+ setString (devName:Collection<String>, value:String, delay:int, tag:String) : void
+ setDouble (devName:String, value:double, delay:int, tag:String) : void
+ setDouble (devName:Collection<String>, value:double, delay:int, tag:String) : void
+ setBoolean (devName:String, value:boolean, delay:int, tag:String) : void
+ setBoolean (devName:Collection<String>, value:boolean, delay:int, tag:String) : void

locationMgr

A Proxy-Object for the LocationManager component. Offers all methods offered by LocationManagers public interface. For details see chapter 8.

7.3. Example-Rules

This section will give a more practical overview on the usage and power of the Rulesystem by presenting a few example-scenarios and their implementation using the Rulesystem described above.

To see these rules in action refer to the tests for this bundle. If not otherwise noted, tests that demonstrate rules similar to the ones presented here can be found in the package `org.fortiss.smartmicrogrid.eventhandler.rulesystem.test.examples`.

7.3.1. Button toggles devices associated to it

Scenario

- Button sends `Toggle`-Events when pressed
- Lights are associated with their respective buttons in the LocationManager
- Lights associated with the button can receive `Toggle`-commands and act accordingly
- **Desired behavior:** Pressing on a button toggles the respective lights

Rule

```
rule "Button Toggle -> device toggle"
when
    ToggleEvent($origin:origin)
then
    commander.toggle(locationMgr.
                        getAssociatedDevices($origin),0,"");
end
```

Listing 2: Button toggles devices associated to it

A similar rule can be found in `DroolsEventHandlerTest.testAddRule()`

7.3.2. Execute an action if the temperature stays below a threshold

Scenario

- Sensor *"enocean?temp1"* sends temperature values as double events with unit "Celsius" when the value changes, every few seconds otherwise.
- When the temperature stays below 25 degrees for 10 seconds (i.e. the maximum sensor value in a 10 second timeframe is below 25), execute some action.
- The rule should only apply to this sensor.

Rule

```
rule "Act if temperature below 25 degrees for 10sec"
when
  $x: Number(doubleValue < 25) from
    accumulate (DoubleEvent($origin:origin == "enocean?temp1",
                            unit=="Celsius", $val: value)
    over window:time(10s),mymax($val))
then
  some.doSomething();
end
```

Listing 3: Rule that matches when temperature stays below 25 degrees for 10 seconds

The part after accumulate (`DoubleEvent(...)`) filters the events to accumulate while `window:time(10s)` specifies the window to look at (`window:length(2)` would look at a window with the last 2 events). `mymax($val)` specifies the value to accumulate (`$val`) and the accumulator to use. `mymax`, the accumulator used here is a custom accumulator that works like the standard maximum-accumulator supplied by drools but adds an important detail: The value returned in case of an empty window (i.e. in case no event was received within the specified timeframe) is clearly defined as the last value received.

This enables the user to write rules consistent with intuition without enforcing a certain frequency of events. Meanwhile, the behavior stays consistent in a more general sense: When no new data was received, the system has to assume the old value is still valid.

Therefore, when no datapoint exists within the current time-frame, the last value is assumed to still be valid and therefore constant within the timeframe, thus the maximum as well.

Of course, more advanced techniques for extrapolating sensor values could also be used but are out of scope for this thesis.

For more information on reasoning over time, see the Drools Fusion Documentation ([6])

Variations

For this rule to apply to all temperature sensors, the check for the origin-value can just be omitted. The rules body can obviously be arbitrary more complicated, e.g. setting values, calling the LocationManager (like seen above).

7.3.3. Evaluating Java-code as condition

Scenario

- Do something when an event from a lightswitch was received.
- Some devices send boolean events but are not lightswitches
- Lightswitches have the property *"devtype"* set to *"lightswitch"* in LocationManager

Rule

```
rule "React to devtype=lightswitch"
when
        ToggleEvent($origin : origin)
        eval("lightswitch"
              .equals(locationMgr.getProperty($origin,"devType")))
then
        some.doSomething();
end
```

Listing 4: Evaluating java code as condition

Note the use of `eval` which takes normal Java-statements that evaluate to a boolean-value which is used when checking the rules condition.

7.3.4. Execute a consequence every 5 seconds

Scenario

- A certain set of java code should be executed every 5 seconds (e.g. send a heartbeat signal)

Rule

```
rule "do something every 5 seconds"
timer (cron:*/5 * * * * ?)
when
then
        some.doSomething();
end
```

Listing 5: Rule to execute a consequence every 5 seconds

Note the use of `timer` after the rule tag (not as condition). Inside the parentheses a standard CRON statement (with additional support for seconds) can be defined with the prefix `cron:`.

For more details on the exact syntax see the Javadoc for the class `org.drools.time.impl.CronExpression`[4] or, more accessible, in the documentation for the Quartz Scheduler [5] from which Drools' implementation was apparently derived.

7.4. Configuration

The Rulesystems configuration is fairly straightforward: Rules are stored in a database and are read from there to be added to the rulebase at startup. Any dynamic changes made via the configuration-interface (see below) are immediately passed on to the Rulesystem itself and are persisted in the database.

7.4.1. Database Schema

Rules are saved in the database without syntactical details (like the `from`, `when`, `end` keywords) as a combination of ID, name, condition and consequence (see figure 5). Condition and consequence contain the repsective parts of the rule in plain text. Name is some plain-text identifier that explains the rules content and makes it easy for users to identify the rules in a graphical configuration-interface or in case of errors. ID is a unique identifier that is used to reference rules when using the configuration-interface.

7.4.2. Configuration Interface

To enable other components (such as a graphical user interface) to configure the Rulesystem, a basic configuration interface (see figure 6) is exposed via the Service Bus(the queue-address is specified in the constant `QUEUE_ADDRESS`. The type `Rule` used within this interface is a container representing one row in the database, i.e. one rule in the Rulesystem.

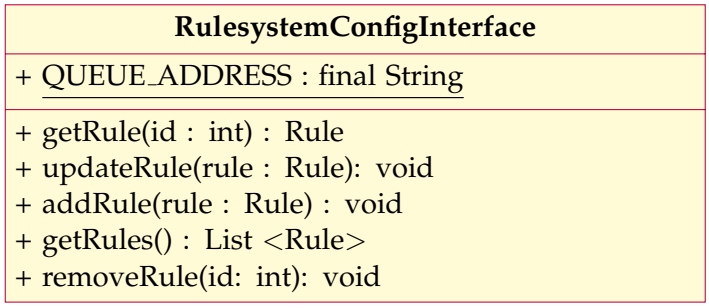
[4] can be found at `http://grepcode.com/file/repository.jboss.org/nexus/content/repositories/releases/org.drools/drools-core/5.1.1/org/drools/time/impl/CronExpression.java` (accessed 19th August 2011)

[5] `http://www.quartz-scheduler.org/documentation/quartz-1.x/tutorials/crontrigger` (accessed 19th August 2011)

Figure 5.: Database Schema for Rulesystem Configuration

Field	Type	
id	`int(11)`	`AUTO_INCREMENT`
name	`varchar(500)`	
cond	`varchar(10000)`	
consequence	`varchar(10000)`	

Figure 6.: The RulesystemConfigInterface



8. LocationManager

Author: Steffen Bauereiß

The LocationManager can easiest be explained as a knowledgebase for the system. It stores information about all devices connected to the system in a database. Assuming the information stored is (due to the name of the component) locations only is wrong. The basic idea of the LocationManager is to store knowledge about associations between devices. An example: one specific lightswitch shall of course toggle one or more defined lights. Both are devices for the system and an association between the lightswitch as a sensor and the light as an actor needs to be stored. This information becomes necessary when talking about rules. As a result of an event triggered by a device connected to the system other devices might have to react. The Rulesystem (chapter 7) finds out whether the system has to react or not. The LocationManager provides the knowledge about which devices have to react upon the event.

8.1. Information stored in LocationManager

The important information stored in the LocationMangager is:

- Device-ID
- Communication-Endpoint
- Associations

Besides this, user-defined properties can be stored for every device as a list of keys and values.

8.1.1. Communication relevant information

The combination of a communication-endpoint and device-ID makes a unique key for accessing other information of the device. At the same time using the communication-endpoint stored for a device the correct wrapper can be determined and commands containing the device-ID can be created and sent.

8.1.2. Control relevant information

The information necessary to determine which devices need to be controlled upon an event is stored as associations between devices. When called the LocationManager provides a list-representation containing all devices associated with a selected device.

8.1.3. User-defined additional information

To improve configurability and UI-Experience or enable a more specific handling of devices in other components, the LocationManager offers a list of key-value-sets for custom properties. These could for example be utilized to store true location-data, more user-friendly device-names, categories like a device-type or other tags. The properties can be accessed in rules and every other component of the system by communicating with the LocationManager and retrieving the properties for a device.

8.2. Data representation

Of course a standard SQL-database can be used to store all the information described above but a graph database offers some benefits like increased performance and easier development due to a fitting data representation. The data representation is suitable because storing device information and associations between devices can exactly be mapped to a graph with devices as nodes and associations between devices as associations between nodes. The implementation of the LocationManager is based on Neo4J [1]. Neo4j includes a key-value store for each node (used for the properties of devices) and offers visualization tools used to easily edit and check what is stored in the databse.

8.3. Implementation

The LocationManagers implementation consists of a database-accessing part and the communication part. The most important functionality of providing a list of associated devices is implemented in a method with this signature:

```
public List<String> getChildren(String parentNodeId,
                                 boolean transitive)
```

It represents the basic idea of the component. All information about devices and their associations is stored in the graph database and whenever a component needs to know which devices have to react upon an event from another device it can retrieve the list of associated devices by calling this method with the device that triggered the event as `parentNodeId`.

A complete list of methods of the LocationManager is shown in the interface definiton of figure 7.

[1] Neo4J The Graph Database `http://neo4j.org/` (accessed 25th September 2011)

Figure 7.: The LocationManagerInterface

LocationMgr
+ QUEUE_ADDRESS : final String
+ addAssociation(parentNodeId : String, childNodeId : String) : void + createNode(nodeId : String): void + getChildren(parentNode : String, transitive:boolean) : List <String> + getPropertyIncludeParent(nodeId : String, key : String) : String + getProperties(nodeId: String) : List <Tuple> + getProperty(nodeId : String, key : String) : String + setProperty(nodeId: String, key: String, value: String) : void + getAssociatedDevices(nodeId: String) : List <String> + getAvailableQueues() : List <String>

9. EnOcean Wrapper

Author: Steffen Bauereiß

9.1. Wrapper - Definition and Description

A wrapper in means of the Micro Grid Living Lab demonstrator as described in section 4.1 is a software component implementing an interface used by every wrapper in order to be able to receive messages from the system. The wrapper usually contains device-specific code for communication with a number of devices that use the same protocol. A wrapper translates from the event/command language used inside the system to a device-specific protocol. This brings an abstractation to the system and makes every device look the same from within the system. Each wrapper has to implement the wrapper interface to ensure commands from the system can be processed by the wrapper. The Interface definition for wrappers is shown in figure 8.

Apart from the WrapperInterface that makes sure methods for receiving commands are available, methods for configuring the wrapper and the devices it controls usually have to be provided as well.

9.2. EnOcean Scenario

As described in section 2.2.1 EnOcean devices are perfectly suitable for demonstration purposes. In order to give a better understanding for the EnOcean-wrapper the following

Figure 8.: The WrapperInterface

WrapperInterface
+ setString(value : String, internalID : String, delay : int, tag: String) : void + toggle(internalID : String, delay : int, tag: String): void + setBoolean(value : boolean, internalID : String, delay : int, tag: String) : void + setDouble(value : double, unit : String, internalID : String, delay : int, tag: String) : void + stopLastCmd(String, internalID : String, delay : int, tag: String) : void + getDevices() : List <Device>

scenario will be used throughout this chapter:

Scenario: A LightSwitch with internal-ID LS is in reach of a Thermokon STC gateway that receives LS's telegrams. An actor connected to a light internally called LA is in reach of the same Thermokon STC. LA is taught to act upon telegrams from the STC containing a channel-ID of 40.

9.3. Device Communication and Telegrams

Devices like the Thermokon STC are capable of receiving EnOcean-telegrams from the devices and passing them to the local network. Telegrams sent to the STC over the network are sent to the EnOcean-devices. The STC can therefore be seen as a communication interface between the Living Lab demonstrator and EnOcean-devices. EnOcean actors and sensors use messages, so-called telegrams for wireless communication. Those telegrams consist of 14 Bytes containing information like a protocol-prefix, a telegram type, data bytes, the senders ID and a checksum (see [3] p. 22ff). Those telegrams are received by the STC that passes them to the EnOcean-wrapper. In reverse direction the wrapper can send telegrams to the STC that passes them to the EnOcean-actors. The STC and EnOcean-wrapper therefore connect the devices to the system.

In the environment of the Living Lab demonstrator each actor controlled by system is taught to understand messages containing the ID of a STC as sender-ID. One STC can control 128 EnOcean-devices by sending a channel-ID within the telegrams ID-bytes.

9.4. EnOcean Wrapper Implementation

Section 9.1 already explains the two sides of a wrapper implementation. When looking at the device-specific part, the EnOcean-wrapper has to connect to the STC in order to be able to communicate with the devices. It has to translate incoming telegrams (from outside the system) into events (used inside the system) and translate incoming commands (from inside the system) into telegrams. The main task of the EnOcean-wrapper, translating between protocols, becomes clear when looking at the simple scenario.

In the scenario the STC would receive a telegram everytime LS gets pressed. This telegram is translated into an event by the EnOcean-wrapper. The event is sent to every EventHandler

Figure 8 shows the wrapper interface implemented by the EnOcean-wrapper. When receiving commands from the system these methods are called. Inside those methods EnOcean-telegrams are created and sent to the STC.

Looking at the scenario the Rulesystem should find a rule to toggle LA when an event from LS is received. Imagine the rule fired and a command to toggle LA is sent to the EnOcean-wrapper. In this case toggle("LA", <int>) is called. The wrapper now creates an outgoing telegram with the information to toggle a light and the id necessary to control the EnOcean-actor LA (40). The telegram is then sent to the STC and at this moment LA gets toggled physically.

The wrapper uses a list of actors and sensors to control mutliple devices. In order to be

Figure 9.: The EnOceanWrapper Configuration Interface

EnOceanConfigInterface
+ getAvailableSensorStrategies(): List < String > + getAvailableActorStrategies(): List < String > + getAvailableWrapperQueueNames(): List < String > + getAvailableChannelCount(queue : String): int + putSensor(queue: String, id : String, strategy : String) : void + putActor(queue: String, channel : int, id : String, strategy : String): void + learn(queue : String, channel : int, data : Strimg) : void + wrapperSetListenMode(queue : String, listening : boolean) : void + getReceivedTelegrams(queue : String) : List < String >

able to control devices some configuration needs to be done inside the wrapper component. For the EnOcean-wrapper there is the so called EnOceanConfigInterface shown in figure 9. Adding a new actor to the wrapper is not only creating a new entry in a list. While this is sufficient for EnOcean-sensors, the actor has to be taught to understand telegrams as described in 2.2.1.

9.5. Strategies

Figure 9 already contains hints towards actor-strategies and sensor-strategies. These strategies become necessary due to the different behaviour of EnOcean-devices. Different sensors send different telegrams with the important information in different representations. Each actor understands another set of commands with again the information inside the telegram in different representations.[1] The strategies help to use one EnOcean-wrapper to control multiple actors and sensors. A suitable strategy needs to be available for each physical device and when added to the system a strategy has to be assigned to the virtual device.

Strategies become more clear when looking at the scenario. In the scenario the wrapper needs a strategy suitable for LA. This strategy should implement at least toggle() and control the light when toggle() is called. Other devices might not need toggle() at all or a toggle() with different code.

[1]The documentation [2] p. 192ff on sensors from “eltako” shows telegram contents for some sensors.

10. UI

Author: Steffen Bauereiß

10.1. General Description

Every IT-system needs some kind of configuration and for the Living Lab demonstrator configuration is an essential issue. Each component needs a very particular kind of configuration depending on what devices it controls and what functionality is implemented. Within the two components Rulesystem and LocationManager the whole logic and environment needs to be set up and each wrapper needs to know which devices it shall control. Controlling these devices is another issue - it may be necessary to access and control the devices directly. This results in a few requirements for a two-sided userinterface described in the following section. The implementation of such a UI can be realised in a variety of ways including native apps for portable devices or any computer-OS and the fully cross-platform option to use websites. This last option is currently used for a first UI-example in the Living Lab demonstator and the details are described in section 10.3.

10.2. UI Principles

When looking at UI principles the systems overall principles can again be considered. The two major principles are to include abstraction and split the UI in two parts:

Abstraction: Where possible common interfaces shall be used to have a unique UI-experience and save on implementing UI-components.

Split in two parts: UI development should be split into two independent parts where possible: *Configuration* and *Control*. This split enables an easier integration of user-access-control later and offers a better differentiation between the functions available.

10.3. Technical Details

In the Service Oriented Architecture of the Living Lab demonstrator any functionality necessary for configuration or control stays within the component itself and is accessed via services. Therefore when talking about UI for the system the only thing that needs to be taken care of is visualising information that is retrieved through those services and giving the user the ability to call the functionalities available. This enables an easy development of UI-components and even the possibility to have multiple UI-components with the same

objective (e.g. one component for configuration through a native Android and another component for configuration through a native iOS-App) running at the same time.

In the first stage of the Living Lab demonstator a web-based UI has been developed. The benefits of cross-platform availability on any operating system and even mobile devices predominated all other factors. In order to support easy access from mobile devices "jQuery mobile"[1] framework was chosen for the frontend.

The backend is implemented using the OPS4J Pax Web framework which allows easy deployment of JSPs and servlets in an OSGi environment. Of special importance is its component *Whiteboard Extender*. It enables the developer to register resources and servlets as standard OSGi services[2] while the framework takes care of all lifecycle management and configuration of the underlying Jetty[3] webserver.

[1] `http://jquerymobile.com/` (accessed 25th September 2011)

[2] for details see `http://team.ops4j.org/wiki/display/paxweb/Whiteboard+Extender` (accessed 25th September 2011)

[3] `http://www.eclipse.org/jetty/` (accessed 25th September 2011)

Part IV.

Further Research and Conclusion

11. Further Research

Author: Florian Sellmayr

Due to the complexity and wide range of the subject, the first architecture prototype for a Smart Micro Grid that is described in this thesis will leave a lot of issues untouched or adressed in only preliminary fashion.

This chapter will point out some of these issues and give some ideas for possible solutions that came up during the work on this project. Please keep in mind that this list is not exhaustive and that ideas presented here are not the result of exhaustive exploration of the subject.

11.1. New Hardware

Author: Steffen Bauereiß

Currently two wrappers are implemented connecting EnOcean-devices and the IPSwitch. The process of adding new hardware and a short description of planned extensions is given in the following sections.

11.1.1. General Workflow for adding new actors and sensors

Adding new actors and sensors to the system can be done in two ways. In the following section no distinction between actors and sensors will be made and the expression "device" shall combine the two categories.

The basic procedure of adding a new device is creating a wrapper for it. Such a wrapper has to implement an interface provided by the system (see figure 8) to make sure communication with the system is guaranteed. Additionally, code for communication with the device has to be implemented and the wrapper needs to send events to the system. The wrappers are implemented as OSGi-Bundle so once implemented the new device can be added to the system by adding it's wrapper (installing and starting the respective bundle) and setting everything up accordingly. An example wrapper-implementation is documented in chapter 9.

The second possibility of adding a new device to the system is reconfiguring an existing wrapper that is implemented in a way to control multiple devices and suitable for the new one. As an example adding new EnOcean-devices is just an issue of adding a new item in the configuration of the already existing EnOcean-wrapper.

11.1.2. Siemens Pac Sentron and Siemens PowerBridge

Siemens' multi-purpose-measurement device Pac Sentron is already listed for future integration to the Living Lab demonstrator. The current idea is to have the Pac Sentron measure power consumption at a few central spots in the building. The device itself can be connected to another device by Siemens called PowerBridge. This device offers the ability to run custom code on it, which in case of the system is predestinated for running the wrapper for the Pac Sentron. The Living Lab demonstrator is a distributed system where a wrapper can run on any device connected to the local network and the Pac Sentron via a PowerBridge is one of first devices to be integrated using this mechanism. Section 11.1.1 describes the general procedure to be applied when integrating these devices.

11.1.3. AC

The Toshiba TCS-Net AC will be integrated into the system in a similar way as the IPSwitch hardware. A wrapper communicating with the web-interface of the AC needs to be added following the general workflow of adding a new wrapper described in section 11.1.1.

11.1.4. Solar Panel

Another project coming up is the integration of a solar-panel into the Living Lab demonstrator with a focus on measurments and some basic control functions. Section 11.1.1 once again describes the general procedure to be used when adding this device.

11.1.5. BACnet

Home automation devices and devices interesting for future versions of the Living Lab demonstrator may happen to use BACnet for communication. With a general BACnet wrapper, any BACnet devices can easily be added to the system. Other protocols used in this field can be addressed in the same or a similar way.

11.2. Reducing complexity for Users

Author: Florian Sellmayr

11.2.1. Rulesystem

As described in chapter 7, the current version of the Rulesystem exposes Drools full power to the end-user, requiring a detailed knowledge of implementation details and the rule-language syntax.

Developing a Domain Specific Language (DSL) for the Rulesystem along with a UI (see section 11.2.2) tailored to it could therefore significantly reduce the complexity of configuration.

Methods to test the effect of rules should also be considered. It could even be advisable to create methods that allow trying out rules on a simulated duplicate of the real system configuration to be able to assess the rules consequence as close to production mode as possible without any negative consequences for the running production system.

11.2.2. UI

During this project, a web-based UI was developed to access and configure all important aspects of the system (see chapter 10).

It was designed to demonstrate the full power of the system and therefore requires quite a lot of knowledge on the systems internal structure and implementation.

Therefore, to make the system more user-friendly to average users, designing a new UI that exposes the necessary functionalities in a simplified fashion is advisable.

It could expose different levels of detail, giving an experienced facility manager access to the full power of the system while normal office employees get to see only a very limited UI that may even be hand-tailored to the specific system environment.

Rulesystem UI

With the rulesystem as of now being the systems most powerful, complex and central component, configuring it easily and reliably is a difficult problem that has not completely been adressed as of now: Only a very basic UI to insert rules exists. Along with the other measures mentioned in section 11.2.1, a hand-tailored UI for this component including an advanced graphical editor for the language could improve the user experience significantly.

11.3. Conflict-Resolution for commands

Author: Florian Sellmayr

In a setup that uses more than one decision-making component (maybe a set of rules combined with a machine learning approach), incoming events can easily trigger conflicting commands.

Detecting and resolving such conflicts while still keeping the system running under the constraints mentioned might prove to be a challenging task with a number of possible algorithmic solutions that were not explored within the scope of this thesis.

The requirement of low latency might make a solution especially hard since it more or less forbids forced delays to wait for other, potentially conflicting commands.

If no feasible algorithmic solution for this problem can be found, introducing “high priority” flags for commands that will not be considered by a delaying conflict resolution algorithm might be necessary.

11.4. Security and Privacy

Author: Florian Sellmayr

While the system is in use only within a limited research and demonstration environment, security is not of primary concern and was not within the scope of this thesis since there are already a number of standard ways to deal with most of these issues.

Nevertheless, when looking at future commercial implementations of such a system, security and privacy are of paramount importance:

Different users should only have certain rights on this system, for example to customize behavior within their own office, not to change the whole buildings behavior.

Security within the architecture could be implemented on several levels, from basic access control at UI-side down to low level authentication and verification of components and users or even full message encryption on SOA-layer. To associate users with a group of devices (e.g. to model a room the user is allowed to control), the existing LocationManager infrastructure could be used.

The system also collects quite a large amount of personal data that could potentially be exploited, for example usage and movement profiles within certain rooms from which an employer could infer the employees daily working hours, the amount of coffee breaks and possibly much more. Thus to make the system acceptable for users and compliant with current privacy laws, additional safe-guards of both technological and organizational nature should be explored.

12. Role-based review

Author: Steffen Bauereiß

As usual the different requirements of multiple stakeholders need to be considered when developing an IT-system. Technical considerations are already listed in chapter 3 but another look at two different stakeholders shall be given here.

The development of the Smart Micro Grid demonstrator has always been driven from two points of view: On the one hand the system has to be configurable, rules and new devices have to be added. On the other hand people are working in smart buildings and of course directly interacting with some of the devices. As a minimum requirement they have to be able to control and to some extent configure devices. Considerations about permissions and user privileges were taken[1] and the high flexibility of the system architecture supports adding control mechanisms for components the user interacts with. Nevertheless the implementation does not include any of these mechanisms so far as they were not in the focus of the development for the first version of the demonstrator. What the implementation does include is a web-interface described in chapter 10 that reflects the two points of view: "Control" and "Configuration". Ideas and considerations behind both parts are topic of the following two chapters.

12.1. Facility Manager

Author: Steffen Bauereiß

A facility manager is most interested in configuring the system. Of course controlling it cannot be ignored but can be skipped for one simple reason: The idea behind a smart building contains a software-component that automatically controls all devices connected. It shall automatically switch to the best power source, control AC and heating, turn off lights after working hours and much more in order to increase energy efficiency. The part of manually controlling the system can be completely moved to the office employee having similar interests. This role is described in section 12.2).

Now with all the background-knowledge given so far, the facility manager in the Living Lab demonstrator needs a configuration interface for the system. He has to add devices and edit rules to get them working. He has to edit connections between devices and wants to keep track of multiple measurements like overall power consumption, open windows after working hours and more.

Basically all components need to support configuration by the facility manager and logging of all data needs to be provided to him.

[1] see section 12.2.3

12.1.1. Logging of Events

Logging is provided by the persistency-component. Each event sent in the system is logged in the database. This will later be extended by an easy way of visualising the data through a graph-generating component and maybe other analysis-components added to the system.

12.1.2. Configuration and initial setup of the system

Configuration interfaces are available for all major components. LocationManager and Rulesystem can be edited through a simple web-interface. The facility manager can set up the basic behaviour of devices by creating rules in the Rulesystem and set the properties and associations for each device in the LocationManger.

A slightly more complex web-interface is provided for the EnOcean-wrapper and will in a similar way be added for future wrappers. The facility manager is responsible for installing new devices like lightswitches and lights. Within a smart building like the Living Lab those need to be added to the system. The different devices often provide different ways of installation depending on the manufacturer (for EnOcean a teach-in-procedure has already been described in 2.2.1). By supporting this installation-procedure at the level of wrappers like e.g. the teach-in-procedure by the EnOcean-wrapper all possibly necessary configuration is provided. This configuration-UI for EnOcean-wrapper is together with the configuration-UI for LocationManager and Rulesystem specifically designed for the facility manager providing easy setup and changing of settings.

12.2. Office Employee

Author: Florian Sellmayr

The counterpart to the facility manager, is obviously a person that is affected by the system as part of his or her environment but not primarily concerned with it. Since the demonstrator is set up in an office environment, the prototype role for this kind of person is typically a normal employee working in one of the offices controlled by the system.

This person will perceive the system as part of the office infrastructure, not as his works primary objective. Most of the time, an office employee will not be interested in tweaking the systems behavior or read detailed data.

Thus, once set up by the facility manager, the system should work well and unobtrusive by default without any additional effort by the office employee. This means, the buttons in the office should be configured in a meaningful way to control the devices inside the office, automatic behavior should work effectively without disturbing the user and sensors should gather the necessary data.

12.2.1. Customizing autonomous behavior

Although default parameters are already set, the office employee might want to customize some parts of the system to his preferences, e.g. set his preferred room temperature, dis-

able behavior he finds to be annoying or stop automatically triggered actions under some circumstances.

Most of these scenarios concern the Rulesystem and similar future components that are responsible for initiating autonomous actions. These components need to offer options to provide a default behavior that can be customized by users within the limited range of their responsibilty or authority (e.g. the users office).

For the Rulesystem, this possibility exists due to the complexity and power of the underlying rule-engine. Rules can be written that utilize additional *"facts"* that can be inserted by users as a way to customize their behavior. It is also possible to allow employees to write additional rules that only apply to their offices and have priority over default rules. At this point however, responsiblity to design such behavior rests solely in the hands of the facility manager since only a generalized interface to the Rulesystem exists that exposes the full power of the Rulesystem without any supporting tools. This was not in the focus of this thesis and is therefore a subject that is still open for further research (see section 11.2.1).

12.2.2. Customizing control

While the facility manager should have configured the controls of the room as best as possible, the office employee might still want to change the way they work[2]. In this case, the employee should be able to make changes to the associations between sensors and actors, i.e. have access to a subset of the LocationManagers configuration.

12.2.3. Security and Access Control

All the scenarios mentioned above require strict and fine grained security and access control mechanisms to make sure the office employees do not, by accident or intentionally, make changes beyond their authority.

Since this is a complex issue that is not directly connected to architecture design (most could be handled isolated at UI level) it was not thoroughly explored in this thesis. For a few ideas, see section 11.4.

[2]The facility manager might have configured the office in a very fine-grained way, giving the employee the chance to control each light individually while the employee always switches all the lights at once.

13. Constraint-based review

Author: Florian Sellmayr

Section 2.3 presented the basic constraints and non-functional requirements for the system. This section will revisit those requirements and match them to the system developed and presented in the previous part of this thesis.

13.1. Flexibility and independence of components

A high flexibility and independence of components was achieved by using the OSGi runtime framework (see section 3.3) and by sticking to a few very generalized interfaces for communication between the main components (see section 5.2).

Thus, new components can easily be added (see section 11.1.1) and components have only (if at all) very limited and clearly defined dependencies between them.

13.2. Scalability

Since communication between components is the most important potential achilles heel of a system that has to integrate such a large number of different components, different approaches have been implemented, reviewed and tested for performance in large scale operation. During this process, the approach presented above proved to be feasible while others were found to have significant downsides (see chapter 6).

13.3. Distributability

Through the use of the ESB-pattern (see section 3.1.2) and message encoding using SOAP (see section 5.1), components can easily be spread over multiple physical locations. The only thing necessary to connect to the system is a network connection to a broker instance.

13.4. Robustness and self-recovery

General guidelines for good programming were followed in the development of components that should lead to robust behavior with respect to faulty input data from other components.

Robustness in the sense of flawed sensor data is mainly an issue for the components receiving events, in our case Persistency and Rulesystem. The issue is of no importance for Persistency since no processing takes place. In case of the Rulesystem, the language

provides methods to deal with such flaws. However, it remains the responsibility of the facility manager designing the rules to use them accordingly.

The lifecycle-management options of OSGi allow the user to easily start, stop or restart single components in case of problems without disturbing any unrelated components. Therefore, the system is prepared for self-recovery mechanisms.

A "watchdog"-component that keeps track of the system-status and initiates such self-recovery options however has not yet been implemented.

13.5. Energy efficiency

Since this project developed only a first technology demonstrator, the system has not been optimized for energy efficiency. Nevertheless, the current systems processing needs are fairly limited, indicating that it could possibly run on low-performance, more energy efficient hardware than the off-the shelf office computers it runs on at this time.

14. Outlook

Author: Steffen Bauereiß

Putting everything together all ideas of a Smart Grid have been given thought when designing the Living Lab demonstrator. On one hand, the system is suitable for controlling a Smart Micro Grid of home automation devices using specifically designed wrappers to connect to them. Adding new hardware is supported and the system can be used for an evaluation of different approaches concering energy efficiency within a Micro Grid. On the other hand, by turning the attention towards extendability and using industry-standard components connectivity is provided by the system. The Living Lab as one participant of a Smart Grid is set up to exchange data with other participants.

Further projects on sharing data, exploring business opportunities, creating a Domain Specific Language, and adding external systems have already been initated and will use the Living Lab demonstrator as a base of their research.

Bibliography

[1] Dirk Slama Dirk Krafzig, Karl Banke. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall PTR, 2004.

[2] Eltako Electronics. The Eltako Wireless system. `http://www.eltako.com/fileadmin/downloads/en/_catalogue/wireless_system_high_res.pdf` (accessed 21th September 2011).

[3] EnOcean. TCM 120 Tranceiver Module - User Manual V1.53. `http://www.enocean.com/de/enocean_module/TCM_120_User_Manual_V1.53_03.pdf` (accessed 21th September 2011).

[4] Richard S. Hall et. al. *OSGi in Action : Creating Modular Applications in Java*. Manning, 2011.

[5] The JBoss Drools Team. Drools Expert User Guide. `http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-expert/html_single/index.html` (accessed 19th August 2011).

[6] The JBoss Drools Team. Drools Fusion User Guide. `http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-fusion/html_single/index.html` (accessed 19th August 2011).